

Programmazione template

Funzionalità ed operatori

- Anche se con comportamenti simili, i vari tipi (`int`, `float`, `double`) in C++ non sono interscambiabili automaticamente
- una medesima azione (es. la somma) deve essere implementata **per ogni tipo** (e **per ogni classe!**)
- il C++ permette di definire operatori con lo **stesso simbolo** per ogni tipo: ad esempio, la somma fra `int`, `float`, `double` si fa con lo stesso operatore `operator+()`: è l'*overloading* degli operatori
- **anche per una classe** si possono ridefinire gli operatori (già fatto con `operator=()`)

Funzioni, tipi, funzionalità

```
double sommaDouble  
(const double & first,  
 const double & second)  
{  
    double somma ;  
    somma = first + second ;  
    return somma ;  
}
```

```
// -----
```

```
int sommaInt  
(const int & first,  
 const int & second)  
{  
    int somma ;  
    somma = first + second ;  
    return somma ;  
}
```

- nonostante la generalizzazione introdotta dall'*overloading* degli operatori, il **problema rimane per le funzioni**
- una medesima funzione (es. la somma, per semplicità) deve essere **implementata per ogni tipo** (e per ogni classe!)
- spesso le funzioni non si basano sulle proprietà del tipo, ma su **alcune sue funzionalità** (`operator+ ()`): si può sfruttare questo fatto!

Generalizziamo: i template

```
template <typename T>  
T somma (const T & first,  
         const T & second)  
{  
    T somma ;  
    somma = first + second ;  
    return somma ;  
}
```

- la funzione è definita sulla base degli operatori di un tipo
- il tipo in questione deve avere gli operatori necessari per la funzione implementati
- nel codice della funzione, “T” prende il posto del tipo in questione
- al momento del *parsing*, il compilatore costruisce l’effettiva implementazione della funzione
- questo significa che le funzioni template vanno implementate nel file “.h”

Come si utilizzano

```
int main (int argc, char** argv)
{
    double d1 = 0. ;
    double d2 = 5. ;

    std::cout << "somma di double: "
               << somma (d1,d2) << std::endl ;

    int i1 = 0 ;
    int i2 = 5 ;

    std::cout << "somma di int: "
               << somma (i1,i2) << std::endl ;

    /* questo da' errore!
    std::cout << "somma mista: "
               << somma (i1,d2) << std::endl ;
    */

    return 0 ;
}
```

- la funzione templata si utilizza come una qualunque funzione nel “.cpp”
- i prototipi effettivi disponibili devono corrispondere rigidamente alla definizione data:
 - siccome nel nostro caso usiamo un tipo “T” per entrambe le variabili da sommare, **non** si possono fare somme “miste”

Due modi d'uso

```
template <typename T>  
T somma (const T& a, const T& b){...}
```

```
int a;  
int b;  
somma<int> ( a, b)
```

```
int a;  
int b;  
somma ( a, b)
```

Il compilatore capisce
che “a” e “b” sono di
tipo “int”

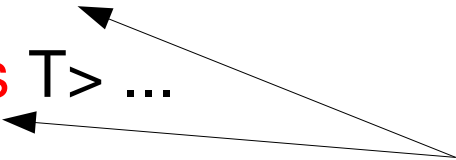
Classi template

- Anche una classe può essere **templata**

template <typename T> ...

template <class T> ...

E' la stessa istruzione!
L'uso di **typename** o **class** qui è
equivalente



- utile per creare “strumenti di lavoro” **più complessi** di quelli offerti dal C++
- esempio: un **vettore** a dimensione variabile (come quello che si crea con `new`) che si distrugge da solo alla fine dell'esecuzione **che possa contenere** `int`, `float`, `double`, `class...`

Una semplice “classe vettore”

```
template <class T>
class vettoreSimple
```

● `template <class T>`

Implementazione
nel file “.h”

```
{
public :
    //! costruttore
    vettoreSimple (const int & elementsNum) :
        m_elementsNum (elementsNum) ,
        m_elements (new T [m_elementsNum])
    {}
    //! distruttore
    ~vettoreSimple ()
    {
        delete [] m_elements ;
    }

    //! ritorna un elemento (modificabile)
    T &
    elemento (const int & i)
    {
        if (i < m_elementsNum) return m_elements[i] ;
        else return m_elements[m_elementsNum-1] ;
    }

private :
    int m_elementsNum ;
    T * m_elements ;
};
```

- Posso usare **T** all'interno della classe
- il numero di elementi viene specificato nel costruttore, che contiene la chiamata a `new`
- il distruttore contiene il `delete`
- il metodo che ritorna ogni elemento controlla il range degli indici

Come si utilizza

```
#include "vettoreSimple.h"
#include <iostream>

int main (int argc, char ** argv)
{
    int elementi = 10 ;

    // definisco un vettoreSimple di interi
    vettoreSimple<int> listaDiInteri (elementi) ;

    // riempio un vettoreSimple di interi
    for (int i=0 ; i<elementi ; ++i)
        listaDiInteri.elemento (i) = i * 2 ;

    // leggo un vettoreSimple di interi
    for (int i=0 ; i<elementi ; ++i)
        std::cout << "elemento " << i
            << " : " << listaDiInteri.elemento (i) << "\n" ;

    return 0 ;
    // il comando "delete" viene chiamato automaticamente
    // nel distruttore della classe vettoreSimple per i vettori
}
```

- la classe è inclusa come al solito
- il tipo da utilizzare per ogni oggetto e' indicato con `<int>`
- l'oggetto si utilizza come un qualunque altro oggetto
- l'elemento del vettore è resituito dal metodo *elemento* (`const int& i`)

Un piccolo *upgrade*

- invece dell'operatore elemento

```
T& elemento (const int& i)
```

si può implementare, per la classe `vettoreSimple`, l'operatore

```
T& operator[] (const int& i)
```

per accedere agli elementi del vettore con le parentesi quadre

Implementazione (file.h)

```
//! ritorna un elemento (modificabile)
T &
operator[] (const int & i)
{
    if (i < m_elementsNum) return m_elements[i] ;
    else return m_elements[m_elementsNum-1] ;
}
```

Utilizzo (file.cpp)

```
// riempio un vettore di interi
for (int i=0 ; i<elementi ; ++i)
    listaDiInteri[i] = i * 2 ;

// leggo un vettore di interi
for (int i=0 ; i<elementi ; ++i)
    std::cout << "elemento " << i
               << " : " << listaDiInteri[i]
               << "\n" ;
```

Esercizi

- **Esercizio 1:** implementare la classe dei vettori e creare in un main un vettore di numeri complessi
- **Esercizio 2:** implementare la classe delle matrici 2x2 e creare in un main una matrice di numeri complessi

Standard Template Library

Che cosa sono

- Classi di **funzionalità generali** (stringhe, insiemi, vettori, mappe associative, funzioni per gestirle) standardizzate
- Utilizzano i **template per essere generali** (come il vettore che abbiamo implementato)
- **Funzionalità garantite**, elastiche, sicure per la memoria (il più possibile)

Reference

Reference of the C++ Language Library, with detailed descriptions of its elements and examples on how to use its functions

The standard C++ library is a collection of functions, constants, classes, objects and templates that extends the C++ language providing basic functionality to perform several tasks, like classes to interact with the operating system, data containers, manipulators to operate with them and algorithms commonly needed.

www.cplusplus.com/reference/

The declarations of the different elements provided by the library are split in several headers that shall be included in the code in order to have access to its components:

algorithm	complex	exception	list	stack
bitset	csetjmp	fstream	locale	stdexcept
cassert	csignal	functional	map	strstream
cctype	cstdarg	io manip	memory	streambuf
cerrno	cstddef	ios	new	string
cfloat	cstdio	iosfwd	numeric	typeinfo
ciso646	cstdlib	iostream	ostream	utility
climits	cstring	istream	queue	valarray
clocale	ctime	iterator	set	vector
cmath	deque	limits	sstream	

std::string

- **Non sono un vettore di caratteri!** Rappresentano una stringa e salvano al proprio interno in maniera ottimizzata le informazioni che ci interessano
- Esistono diversi **operatori** che si possono utilizzare sulle stringhe
- Per recuperare una stringa alla maniera del C si utilizza il metodo `std::string::c_str ()`

Esempio veloce

```
// creo e riempio una stringa  
std::string parola = "esempio" ;  
std::cout << parola << std::endl ;
```

- #include <string>
- **Creo** una stringa e le assegno un valore

```
// creo una stringa a partire da altre stringhe  
std::string frase = altraParola + " " + parola ;  
frase += "!" ;  
std::cout << frase << std::endl ;
```

- **Creo una stringa a partire da un'altra stringa**
- Si può utilizzare l'operatore `operator+ ()`, che concatena le stringhe

std::vector

- Ha la **stessa funzione di un *array*** allocato dinamicamente (come il nostro vettore)
- La gestione della memoria è **ottimizzata e sicura**
- **Si riempie** con il metodo `std::vector::push_back ()`
- Copia gli oggetti il meno possibile, **ma lo fa**, quindi è necessario che sia **ben definito il *copy constructor* e l'operatore *operator= ()*** degli oggetti
- **Si rilegge con gli iteratori**, che sono oggetti che si comportano in modo simile ai puntatori
- Si può leggere anche alla maniera tradizionale del C

Qualche esempio

```
// definisce un vector
std::vector<double> vectorDiReali ;

int quanti = 10 ;
// riempie il vector
for (int i=0 ; i<quanti ; ++i)
    vectorDiReali.push_back (i * 3.14) ;

// si puo' usare come un vettore "solito"
for (int i=0 ; i<quanti ; ++i)
{
    std::cout << "elemento " << i
                << ": " << vectorDiReali[i] << "\n" ;
}
```

```
vectorDiReale.at (i)
```

- `#include <vector>`
- Il vettore viene **creato** templato sul tipo che contiene, e lui stesso è un oggetto (quindi ha i suoi metodi ed i suoi membri)
- Per **riempirlo** si usa il metodo `push_back ()` senza preoccuparsi delle dimensioni (fa tutto lui)
- Si può **rileggere come un normale array del C** (anche l'uso della memoria è analogo, in entrambi i casi si utilizzano regioni contigue della memoria)
- Il metodo `at ()` sostituisce `[]`

Uso degli iteratori

- Il modo di **accedere ai** `vector` **elemento per elemento**, sia per leggerlo che per modificarlo in un loop

```
// legge il vector: l'iteratore e' costante, non posso fare modifiche
for (std::vector<double>::const_iterator itVec = vectorDiReali.begin () ;
     itVec != vectorDiReali.end () ;
     ++itVec)
{
    std::cout << "elemento "
               << itVec - vectorDiReali.begin ()
               << ": " << *itVec << "\n" ;
}
```

```
// modifica il vector: l'iteratore non e' piu' costante
for (std::vector<double>::iterator itVec = vectorDiReali.begin () ;
     itVec != vectorDiReali.end () ;
     ++itVec)
{
    *itVec += 2 ;
}
```

- Il vettore restituisce il **proprio inizio** (`begin ()`) e un **elemento oltre la fine** (`end ()`)

Nel ciclo, l'iteratore si usa **come un puntatore**

Ci sono **due tipi di iteratori**: quello costante (solo per leggere) e quello non costante (anche per modificare)

std::map

- Le STL non hanno soltanto i `vector`, ma una **serie di container per diverse funzioni**
- Una **mappa associativa** è una generalizzazione dell'elenco del telefono:
 - c'è una **chiave di ricerca** (nome)
 - c'è una **informazione** (numero)
- Come per i `vector`, c'è il modo di **riempirla e rileggerla** con i suoi stessi metodi ed operatori
- Una mappa può contenere **un solo elemento per ogni chiave**
- Una mappa è un insieme **ordinato secondo la chiave**

Creare una mappa

```
// definisce una mappa  
std::map<int,double> mapDiReali ;
```

```
int quanti = 10 ;  
// riempie la mappa  
for (int i=0 ; i<quanti ; ++i)  
    mapDiReali[2*i] = (2 * i * 3.14) ;
```

- `#include <map>`
- **Templata** sulla **chiave** (primo tipo) e **oggetto** (secondo tipo)
- **La chiave deve essere ordinabile**, quindi è necessario che sia definito un `operator< ()` per l'oggetto per la chiave (minore stretto!)
- La mappa viene riempita con un operatore di assegnazione; anche in questo caso, ***copy constructor*** e `operator= ()` devono essere definiti con attenzione

Leggere o modificare una mappa

```
// legge la mappa: l'iteratore e' costante, non posso fare modifiche
for (std::map<int,double>::const_iterator itMap = mapDiReali.begin () ;
    itMap != mapDiReali.end () ;
    ++itMap)
{
    std::cout << "elemento "
              << itMap->first
              << ": " << itMap->second << "\n" ;
}

// modifica il map: l'iteratore non e' piu' costante
for (std::map<int,double>::iterator itMap = mapDiReali.begin () ;
    itMap != mapDiReali.end () ;
    ++itMap)
{
    // itMap->first *= 3 ; // questo non puo' essere modificato
    itMap->second *= 3 ;
}
```

- Per leggere la mappa uso di nuovo **una classe iteratore** (diverso da quello dei `vector`!)
 - L'iteratore nel ciclo si comporta **come un puntatore**
 - **La chiave e l'oggetto** della mappa si recuperano con due membri: **first** e **second** rispettivamente
 - Esistono iteratori **costanti** (sola lettura) e **non costanti** (anche modifica)
- **La chiave non può essere modificata**, per non sovvertire l'ordinamento (cosa che corromperebbe il container)
 - Per eliminare un elemento della mappa -> `map.erase(iteratore);`

Riassumendo

- Le Standard Template Library offrono una vasta gamma di **container e algoritmi utili e sicuri**
- Si utilizzano librerie scritte da altri (esperti!) per **semplificare la nostra vita**
- Le STL sono **standardizzate** dalla Standardization Committee, quindi sono portabili
- Le STL sono uno strumento molto potente, noi abbiamo visto soltanto **la punta della punta dell'iceberg**
- **Non si fa C++ senza**

Esercizi

- **Esercizio 3:** crate una mappa simil-elenco del telefono e riempitela inserendo come chiave in nomi (string) e come oggetti i numeri di telefono.
- **Esercizio 4:** creare un `vector` di istogrammi che disegnino due o più distribuzioni casuali, che seguono diverse pdf (con quanto fatto all'inizio)
- **Esercizio 5:** creare una mappa di istogrammi associati ciascuno ad una stringa, che identifica la funzione utilizzata per disegnarli

Approfondimenti: i namespace

- I namespace permettono di raggruppare oggetti con un nome

```
namespace spazio {  
    int a;  
    double b;  
}
```

- Per utilizzarli

```
spazio::a  
spazio::b
```

- Proprio come il namespace “std”

```
std::cout << ciao << std::endl;
```

- L'uso di namespace è safe e il codice è di facile debugging.
- Si può comunque usare ...

```
using namespace std;
```


Cicli automatici

```
#include <algorithm>
#include <functional>
```

```
struct stampa:
    public std::unary_function<double, void>
{
    void operator() (double& x)
    {
        std::cout << x << "\n" ;
    }
};
```

```
for_each (vectorDiReali.begin (),
          vectorDiReali.end (),
          stampa ());
```

- In un ciclo in cui **le stesse operazioni** vengono eseguite su ogni elemento di un `vector` si puo' **definire la funzione** e lasciare che sia un algoritmo esistente a fare il ciclo nel modo migliore
- La funzione e' definita secondo un **protocollo standard** (`unary_function`), che sta nell'header `<functional>` ed è templata sugli input ed output che vogliamo che abbia
- **l'algoritmo** che fa il ciclo (`for_each`) sta nell'header `<algorithm>`: l'algoritmo prende in ingresso i limiti del vettore e la funzione da applicare per ogni suo elemento

L'ordinamento dei vector

```
sort (vectorDiReali.begin (),  
      vectorDiReali.end ());
```

```
struct mySort:  
public std::binary_function<int, int, bool>  
{  
    bool operator() (int x, int y)  
    {  
        return x < y ;  
    }  
};
```

```
sort (vectorDiReali.begin (),  
      vectorDiReali.end (),  
      mySort ());
```

- L'algoritmo `for_each` agisce su ogni elemento del vettore **indipendentemente dagli altri**
- Un esempio di operazione che riguarda non solo i singoli elementi, ma anche le relazioni fra di loro, è **l'ordinamento di un vector**
- **L'algoritmo `sort`** ordina un vettore: il **comportamento di default** è basato sull'`operator< ()`, implementato per l'oggetto contenuto nel vettore
- Si può definire un **operatore di minore fatto in casa** e passarlo all'algoritmo `sort` per ordinare il vettore in modo alternativo (ad esempio, se in un vettore di interi si volessero prima i pari, poi i dispari)